Comparison between internal and external DSLs via RubyTL and Gra2MoL

Jesús Sánchez Cuadrado, Javier Cánovas, Jesus Garcia Molina

Comparison between internal and external DSLs via RubyTL and Gra2MoL

Jesús Sánchez Cuadrado (jesus.sanchez.cuadrado@uam.es)
Universidad Autónoma de Madrid, Spain
Javier Luis Cánovas Izquierdo (javier.canovas@inria.fr)
AtlanMod, École des Mines de Nantes – INRIA – LINA, France
Jesús García Molina (jmolina@um.es)
Universidad de Murcia, Spain

ABSTRACT
Domain Specific Languages (DSL) are becoming increasingly more important with the emergence of Model-Driven paradigms. Most literature on DSLs is focused on describing particular languages, and there is still a lack of works that compare different approaches or carry out empirical studies regarding the construction or usage of DSLs. Several design choices must be made when building a DSL, but one important question is whether the DSL will be external or internal, since this affects the other aspects of the language. This chapter aims to provide developers confronting the internal-external dichotomy with guidance, through a comparison of the RubyTL and Gra2MoL model transformations languages, which have been built as an internal DSL and an external DSL, respectively. Both languages will first be introduced, and certain implementation issues will be discussed. The two languages will then be compared, and the advantages and disadvantages of each approach will be shown. Finally, some of the lessons learned will be presented.

INTRODUCTION
Software applications are normally written for a particular activity area or problem domain. When building software, developers have to confront the semantic gap between the problem domain and the conceptual framework provided by the software language used to implement the solution. They must express a solution based on domain concepts using the constructs of a general purpose programming language (GPL), such as Java or C#, which typically leads to repetitive and error prone code. This encoding task is considered to be “not very creative, and more or less waste or time”, and existing code maintenance is difficult (Dmitriev, 2004). Since the early days of programming, domain-specific languages (DSLs) have therefore been created as an alternative to using GPLs.

DSLs allow solutions to be specified by using concepts of the problem domain, thus reducing the semantic gap between them, and thereby improving productivity and facilitating maintenance, as a number of studies and case studies report (Weiss & Lai, 1999; Ledeczi, Bakay, Maroti, Volgyesi, Nordstrom, Sprinkle &Karsai, 2001; Kelly & Tolvanen, 2008; Kosar, Mernik & Carver, 2011). DSLs are not new (Bentley, 1986), for instance SQL, Pic or Make are well-known examples, but the interest in them has increased considerably in the last decade with the emergence of *model-driven development* paradigms (“MDA Guide”, 2001; Kelly & Tolvanen, 2008; Greenfield, Short, Cook & Kent, 2004; Voelter, 2008), which provide systematic frameworks for the building and use of DSLs, their core being meta-modeling.

Model-driven paradigms are based on three basic principles. Firstly, a software application is partially (or totally) described using models, which are high-level abstract specifications, rather than using solely a GPL. Secondly, these models are expressed with DSLs which are created by applying meta-modeling (i.e. the DSL abstract syntax is represented as a meta-model). Thirdly, automation is achieved by means of model transformations which are able to directly or indirectly transform models (e.g., DSL programs) into the final code of the application by creating intermediate models. Two kinds of model transformation languages are therefore needed (Czarnecki & Helsen, 2006): model-to-model transformation languages, which allow us to express how models are mapped into models, and model-to-text transformation languages, which allow us to express how models are mapped into text (e.g., GPL code). Model-based techniques can also be applied in software modernization tasks, and a third kind of model transformation with which to extract models from legacy software artifacts (e.g., GPL code or a XML document) is then involved, which is normally called text-to-model transformation.

A DSL normally consists of three basic elements: abstract syntax, concrete syntax, and semantics. The abstract syntax expresses the construction rules of the DSL without notational details, that is, the constructs of the DSL and their relationships. Meta-modeling provides a good foundation for this component, but other formalisms such as BNF have also been used over the years. The concrete syntax defines the notation of the DSL, which is normally textual or graphical (or a combination of both). There are several approaches for the semantics (Kleppe, 2008), but it is typically provided by building a translator to another language (i.e., a compiler) or an interpreter.

Several techniques have been proposed for the implementation of both textual DSLs (Fowler, 2010; Mernik, Heering & Sloane, 2005) and graphical DSLs (Kelly & Tolvanen, 2008; Cook, Jones, Kent & Wills, 2007). In this work we focus on textual DSLs, and particularly consider two kinds or styles according to the implementation technique used: external DSLs and internal DSLs. An external DSL is typically built by creating a parser that recognizes the language's concrete syntax, and then developing an execution infrastructure if necessary. An internal DSL, however, is implemented on top of a general purpose language (the host language), and reuses its infrastructure (e.g., concrete syntax, type system and run-time system), which is extended with domain specific constructs. The DSL is therefore defined using the abstractions provided by the host language itself. For instance, in an object-oriented language, method calls can be used to represent keywords of the language. Languages with a non-intrusive syntax (e.g., LISP, Smalltalk or Ruby) are well suited for use as host languages.

A number of design decisions must be made when building a DSL, such as those related to its concrete syntax, how the language semantics is going to be defined and in which form (interpreted or compiled), or whether there will be an underlying abstract syntax. However, deciding whether the DSL will be internal or external will have an impact on the other aspects of the language. Making an effective choice between these two options therefore requires a careful evaluation of the pros and cons of each alternative. Some important aspects that should be evaluated are the following, which are related to the three elements of a DSL: abstract and concrete syntaxes, and semantics (executability and optimizations), and to quality criteria (extensibility and efficiency) and DSL tooling (tools for developing DSL and tools for using DSL).

- **Concrete syntax**. Does the DSL require a specialized syntax? Is the host language syntax suitable for the DSL? How much effort is needed to embed the DSL in comparison to building the DSL from scratch?
- **Abstract syntax**. In which cases might an abstract syntax be necessary, and in which is it possible to manage without it? How different is it to support an abstract syntax in each case? This last issue is related to the following aspect.

- **Executability**. How much does the host language assist in the executability of the DSL? Do we need to adapt the (internal) DSL to facilitate its executability? In which cases is it most recommended to create an interpreted/compiled language?
- **Optimizations**. Can the execution process be optimized to improve the efficiency?
- **Language extension**. How difficult is it to incorporate new constructs into the language?
- **Integration and library availability**. How can an internal/external language facilitate integration with other tools such as editors? Are the libraries required available in the chosen host language?
- **DSL development tools**. Are there tools that facilitate the creation of internal/external DSLs? How much freedom do they offer in the creation of the language? Do these tools support the aspects identified in this comparison?
- **Target audience and usability**. Does the target audience expect a language with a special syntax? Are they already used to the host language syntax?

Over the last few years we have gained some experience in developing DSLs for model-driven environments. Some of these have been built as internal DSLs (Sanchez Cuadrado & García Molina, 2007) and others as external DSLs (Diaz, Puente, Cánovas & García Molina, 2011; Cánovas & García Molina, 2009). Notably, we have developed RubyTL (Sánchez Cuadrado, García Molina & Menárguez, 2006) and Gra2MoL (Cánovas & García Molina, 2009) as internal and external DSLs, respectively. RubyTL is focused on model-to-model transformations, while Gra2MoL is intended to perform text-to-model transformations that are typically needed in modernization projects to obtain a model-based representation of source artifacts that are described by a grammar. Although each language is focused on addressing a specific MDE task, they share two characteristics: both are transformation languages (model-to-model and text-to-model, respectively), and both are inspired by the ATL transformation language (Jouault, Allilaire, Bézivin, & Kurtev, 2008) since both rely on rule and binding concepts as their main constructs, signifying that their execution mechanisms are alike. In addition, both languages have a navigation language, but of a different nature in each case.

As noted in (Mernik, Heering & Sloane, 2005) there is a shortage of guidelines and experience reports on DSL development. This chapter aims to provide guidance when confronting the external vs. internal dichotomy by discussing the design decisions involved in the creation of RubyTL and Gra2MoL. As both languages share similar features they provide a case study with which to compare both approaches, and what is more, to observe the results (benefits and drawbacks) of each approach. Both languages are first introduced, along with an explanation of their commonalities and differences, and some particular requirements that they have to satisfy. The aspects mentioned above are then discussed in the light of RubyTL and GraMoL, and finally some lessons learned are presented.

RUBYTL – AN INTERNAL DSL FOR MODEL-TO-MODEL TRANSFORMATION

RubyTL was created in 2005, as part of a project initiated to experiment with model transformation language features. To this end we planned to build an extensible model transformation language in order to gain some experience in model transformation languages and devise new features. A rapid and flexible implementation was therefore needed, and this was the main factor involved in our decision to implement this language as an internal DSL in Ruby.

During the first versions of RubyTL, several extensions were implemented, and their usefulness was tested by building transformations that put them in practice. The language later proved to be useful as a normal transformation language, and not only for experimentation purposes, so some of the extensions

were selected and added to the stable version of the language. RubyTL will now be introduced by means of an example, and some implementation notes are then provided.

Language description

RubyTL is a hybrid model-to-model transformation language, meaning that it provides both declarative and imperative constructs with which to write transformation definitions. The declarative part is inspired by ATL, which is based on the rule and binding concepts. Rules establish mappings between a source meta-model type and a target meta-model type, while a binding is a special kind of assignment that establishes a correspondence between a source type feature and a target type feature. As will be shown, a binding is resolved by a rule. Interestingly, the imperative part of RubyTL is reused from Ruby for free (i.e., any Ruby construct is valid in RubyTL).

Figure 1. RubyTL Transformation example. (a) Source Java metamodel. (b) Excerpt of the UML metamodel considered in the example. (c) RubyTL transformation definition.

We shall illustrate the language by using an example that transforms Java code represented as a model into a UML model. Figure 1a shows the source Java meta-model, while Figure 1b shows an excerpt of the target UML meta-model. The source meta-model represents Java classes (`Class` metaclass) along with their methods (`methods` reference) and fields (`fields` reference), while the target meta-model represents UML classes (`Class` metaclass) and their properties (`ownedAttribute` reference). The piece of code listed in Figure 1c is the RubyTL model transformation for this example, in which every Java class is transformed into a UML class, and whenever such a class contains a `getInstance` method, it is considered as a singleton class. Java class fields are additionally transformed into UML class properties, and a property is marked as read-only when there is no method in the class, following the Java convention for setting attributes.

The transformation has two rules, `javaclass2class` and `field2property`. As can be seen, a rule has a `from` part in which the source element metaclass is specified, a `to` part in which the target element metaclass is specified, and a `mapping` part in which the relationships between the source and target model elements are specified. These relationships are expressed either in a declarative style through of a set of bindings or in an imperative style using Ruby constructs. It is worth noting that both bindings and Ruby constructs can be mixed.

In the example, the first rule is of “top” type, signifying that it is applied to each instance of `Java::Class`. Applying a rule means creating the target element metaclass and executing its mapping part. The second rule will be executed lazily, in the sense that it will be invoked only if it is needed to resolve a binding.

A binding is an assignment in the form *target.property = source-expression* where *source-expression* is a Ruby expression whose result is either an element or a collection of elements belonging to the source model. When a binding is evaluated, if the right-hand side type is different from the left-hand side type, a rule whose source type (`from` part) conforms to the right-hand type and whose target type (`to` part) conforms to the left-hand type is looked up. If found, the rule is applied using the right-hand side element of the binding as the source element, and the target element obtained is assigned to the target property. For example, the `uml_class.ownedAttribute = java_class.fields` (line 6) binding is resolved with the `field2property` rule.

As can be seen, it is possible to write imperative code in a rule (lines 7-9) using the regular Ruby syntax. In this respect, all of Ruby’s features and libraries are available. For instance, lines 7 and 19 make use of the Ruby collection library to navigate models in an OCL-like style, and line 20 uses built-in regular

expressions. It is worth noting that these features are provided free because RubyTL is a Ruby internal DSL, and provides developers with a means to tackle complex transformations when the declarative style is not sufficient.

Implementation issues

The specific techniques used to implement an internal DSL depend on the paradigm the host language belongs to. In this case, as Ruby is a dynamic object-oriented language, the aspects commented on as follows are more amenable to be applicable to this kind of languages.

At the concrete syntax level, the basic implementation technique was to identify the language keywords (e.g., `rule`, `from`, `to`) and to map each keyword into a method, with zero or more parameters. For instance, the `rule` "keyword-method" takes the name of the rule as a parameter. A nested structure of the language was also mapped into a code block, which was passed as an implicit parameter to the corresponding keyword-method. For instance, the elements of a rule (`from`, `to`, `mapping`) are enclosed within a code block (`do – end`), which would be a second parameter of the `rule` keyword-method. Precise details of this technique can be found in (Sánchez Cuadrado & García Molina, 2009).

An internal DSL may use an underlying abstract syntax model, which is created as a result of evaluating the keyword-methods, and this is in some way evaluated afterwards. In (Fowler, 2010), the creation of this semantic model is considered essential if well-designed DSLs are to be obtained. The alternative would be to perform actions while the keyword-methods are evaluated (much in the style of syntax-directed translation (Aho & Ullman, 1977)). RubyTL uses a mixed approach, in which an abstract syntax model is obtained while keyword-methods are being evaluated, but with the distinguishing feature that, for the *mapping* keyword, the corresponding code block is captured to be executed later, as we shall explain below. This is done by allowing the abstract syntax of RubyTL point to the runtime Ruby object which represents the mapping code block.

With regard to the execution strategy, the transformation definition (represented by its abstract syntax model) is evaluated by the RubyTL interpreter. In fact, the classes that represent the abstract syntax include methods with which to perform the evaluation. For instance, in order to start the transformation, a method called `execute_at_top_level` is called for each top rule object. It applies the rule to all instances of the corresponding source type specified in its *from* part, thus creating an instance of the target type specified in the *to* part and executing the *mapping* part which has been captured as a code block. It is interesting to note that the execution of the code block is left to the Ruby interpreter. Since the content of the code block is out of the control of the RubyTL interpreter (i.e., it is regular Ruby code), the effect that a rule is invoked in order to resolve a binding is thus achieved by overloading the assignment operator in such a way as to search for the correct rule to transform the right-hand side part of the binding assignment into the left-hand side part. This technique makes it possible to leave the evaluation of expressions and imperative code to the Ruby interpreter, while keeping the transformation algorithm under control.

Finally, an important concern if a model transformation language is to become mainstream is its interoperability with other tools. At the time of developing RubyTL, Ecore/EMF was (and still is) the most frequently used meta-modeling framework, but it is written in Java, which hindered its use with RubyTL. Thus, when RubyTL began to be used by developers outside our team, interoperability became more important, and we therefore had to create an Ecore-compatible framework in Ruby. To this end we joined the RMOF project (http://rmof.rubyforge.org), and integrated RMOF with RubyTL to achieve interoperability with Ecore/EMF.

GRA2MOL – AN EXTERNAL DSL FOR TEXT-TO-MODEL TRANSFORMATION

In the context of a Struts-to-JSF migration project back in 2007, we needed to obtain models from some existing Java code in order to apply a model-driven modernization process. Extracting models from GPL source code requires establishing a mapping between elements of a grammar and elements of a target meta-model. Implementing this mapping involves intensive tree traversals in order to resolve references, that is, transforming the identifier-based implicit references between elements of the syntax tree into explicit references between model elements. Bearing in mind our previous experience, we decided to build a DSL, called Gra2MoL, which was tailored to the model extraction problem as an alternative to implementing ad-hoc parsers. The two main choices in the design of Gra2MoL were: providing a query language that was specially adapted to traverse and retrieve information from syntax trees, and allowing the grammar–meta-model bridge to be expressed in a RubyTL-style. Gra2MoL is in fact a text-to-model transformation language with which to extract models from any kind of artifact conforming to a grammar. To the best of our knowledge, Gra2MoL is the first language that uses a rule-based transformation approach for this type of problems.

From our experience in developing RubyTL, we decided to implement Gra2MoL as an external DSL owing to the complexity of the query language and scalability concerns, as will be commented on later. Gra2MoL and its query language will now be introduced by means of an example, and some implementation issues are then commented on.

Language description

Gra2MoL is a rule-based transformation language in the style of RubyTL, but with two important differences: i) the source element of a rule is a grammar element rather than a meta-model element and ii) the navigation through the source code is expressed by a query language that is specific to syntax trees, rather than an OCL-like language (which would require writing complex and large navigation chains, since its objective is to traverse regular models).

Throughout this section we shall use an example in order to illustrate the syntax and semantics of the language. The example could be part of model-driven modernization process of a Java system, where the first step would be to obtain Java models from Java source code. These Java models could later be the input of a model-to-model transformation, such as that presented previously for RubyTL.

Figure 2a shows an excerpt of the Java grammar considered in the example, whereas the Java meta-model has already been presented in Figure 1a. The grammar represents classes (`classDeclaration` rule) and their corresponding bodies (`classBody` rule), which can include several declarations (`memberDecl` rule). In this example we shall deal solely with method declarations (`methodDeclaration` rule). The corresponding Gra2MoL transformation in this example is composed of two rules, which are shown in Figure 2b.

Figure 2. Gra2MoL transformation example. (a) Excerpt of the Java grammar considered in this example. (b) Gra2MoL transformation definition.

As can be observed, a Gra2MoL rule has a structure which is very similar to that defined for RubyTL. A Gra2MoL rule is composed of `from`, `to`, `queries` and `mappings` parts. The `from` part specifies the source grammar element and declares a variable that will be bound to a syntax node element when the rule is applied. The `to` part specifies the target type. The `queries` part contains a set of query expressions that allow information to be retrieved from the syntax tree representing the source code. Finally, the `mappings` part contains a set of bindings with which to initialize the features of the target meta-model element. Unlike RubyTL, in which Ruby imperative code can be written along with the bindings, in Gra2MoL only binding constructs can be used in this part.

Like the RubyTL example, the first rule of the example is of the “top” type, which means that it is executed for every element of its *from* type, and its bindings will yield the execution of other rules. The execution of a Gra2MoL transformation is therefore also driven by the bindings, whose syntax and semantics are similar to those previously explained for RubyTL. In Gra2MoL, the type of the source expression can be a literal value or one or more syntax tree elements. Like RubyTL, when a binding is evaluated, if the right-hand side type is different from the left-hand side type, a rule whose source type (`from` part) conforms to the right-hand type and whose target type (`to` part) conforms to the left-hand type is looked up. Whenever a rule is found, it is applied to the right-hand side element of the binding as a source element, and the target element obtained is assigned to the left-hand side property.

In the example, the `createClass` rule starts the transformation. This rule defines the mapping between the `classDeclaration` grammar element and the `Class` metaclass, that is, it creates an instance of the `Class` metaclass from every `classDeclaration` node in the syntax tree representing the source code. The `queries` part of the rule includes one query which collects all the method declarations of the class. The syntax and semantics of the query language will be explained in the following section. On the other hand, the mappings part of the `createClass` rule initializes the features of the target element. The first mapping sets the name attribute with the value obtained from accessing the `classId` leaf of the tree node matched by the rule (`cd` variable). The second binding, whose right-hand side part is the result of the `ms` query, is resolved by looking up and executing a conforming rule. In this case, the conforming rule is the `createMethod` rule, which is executed for each result node of the `ms` query, and it only assigns the `ID` (a leaf node with a string value) to the name of the method. The element created by the rule will be added to the `methods` reference.

The query language

One distinguishing feature of Gra2MoL is its structure-shy language inspired by XPath (XPath, 2011). It is tailored to navigate syntax trees in as simple manner, thus avoiding the need to define every navigation step by using XPath-like operators.

A query in Gra2MoL consists of a sequence of query operations, each of which includes four elements: an operator, a node type, a filter expression (optional) and an access expression (optional). There are three types of operators: `/`, `//` and `///`. The `/` operator returns the immediate children of a node and is similar to dot-notation (e.g., in OCL). The `//` and `///` operators permit the traversal of all the node children (direct and indirect), thus retrieving all nodes of a given type. The `///` operator searches the syntax tree in a recursive manner, whereas the // operator only matches the nodes whose depth is less than or equal to the depth of the first matched node. These two operators allow us to ignore intermediate superfluous nodes, thus making the query definition easier, since it specifies what kind of node must be matched, but not how to reach it, in a structure-shy manner. The `#` operator is used to indicate the type of root nodes of the query result and must be associated with one and only one query operation.

As an example, the rule `createClass` uses the query `/cd//#methodDeclaration`, which collects all the `methodDeclaration`’s children (direct and indirect) of the node represented by the `cd` variable, which is a `classDeclaration` node. The same query expressed in the expression language provided by RubyTL (i.e., an OCL-like language) is as follows:

```
# Given a node “ClassDeclaration” namedncd
ifncd.classBody.classBodyDeclaration == nil
   []
else
  ncd.classBody.classBodyDeclaration.
    select { |decl| decl.kind_of?(MemberDcl) }.
    select {|member| member.kind_of?(MemberDeclaration) }
end
```

It is worth noting how the clarity, legibility and conciseness are improved, because this query language is better suited to this domain (text-to-model transformation) than an OCL-like language (which is more general) like that provided by RubyTL.

Implementation issues

As previously explained, although RubyTL and Gra2MoL have a similar syntax, the latter was implemented as an external DSL, principally to facilitate the implementation of the query language and to improve the scalability. The concrete syntax of the language was therefore defined with a grammar (from which we built the parser of the language), thus allowing us to tune the syntax more easily.

Gra2MoL uses abstract syntax models to represent transformation definitions. These models were initially obtained by using an ANTLR-based parser with annotations (i.e., actions) in the language grammar. Once a first prototype of the language had been obtained, we defined a kind of bootstrap process with which to obtain abstract syntax models from textual transformation definitions, since Gra2MoL can actually be used to extract models from any text conforming to a grammar. This process receives as inputs the grammar of the language, the abstract syntax meta-model, the transformation definition and the Gra2MoL transformation definition, and outputs are the abstract syntax model corresponding to the transformation definition of interest.

Regarding the language execution, the Gra2MoL engine executes transformation definitions in three phases. In the first phase, the source grammar is automatically annotated in order to generate a parser that is able to create a concrete syntax tree from the source code. This syntax tree is later used to execute the queries. In the second phase, the bootstrap process obtains the abstract syntax model from the textual transformation definition, as indicated above. This model is eventually used in the third phase by an interpreter that executes the transformation rules. While the rules are applied, the queries are also interpreted and executed through the use of the syntax tree obtained in the first phase. As a result of the transformation execution, the interpreter generates the model extracted from the source code according to the transformation rules.

Besides the execution engine, we also developed an Eclipse plug-in that incorporates some development tools which facilitate the definition of new transformations (e.g., language-specific text editor with syntax highlighting and formatting, outline view, etc).

The language also incorporates an extension mechanism which allows new operators to be added to the rules, particularly in the queries and mappings parts. When developing new operators, it is necessary to provide both their functionality and a simple syntax. Since Gra2MoL has been developed in Java, the functionality of new operators must be implemented by using the extension framework provided by the language. With regard to the syntax for the new operators, in order to avoid having to modify the grammar of Gra2MoL for each new extension, the *ext* keyword allows the new operator to be referenced by name. For instance, if `digestName` is an extension that deals with string values, it is possible to write `name = ext digestName(“some name”)`.

COMPARISON OF RUBYTL AND GRA2MOL

Deciding whether to create a language as an internal or an external DSL is a key decision since it affects the other decisions involved in the process of creating the DSL, and more importantly, it may determine its success. This is for several reasons. First of all, the freedom to define the desired concrete syntax is very different in each case. Secondly, as we shall comment on later, the user perception of the DSL is typically different when it is internal or external. Finally, once the decision has been made, it is not easy to change to the other option since the implementation of most components of the languages is dependent on choice.

As mentioned previously, during the last few years we have gained some experience in developing both internal and external DSLs, learning a few lessons along the way. In this section we compare RubyTL and Gra2MoL with the series of aspects presented in the introductory section. These aspects should be taken into account in order to make an informed choice, based on the knowledge of the trade-offs of each approach with regard to the problem that is being addressed. The decisions made when building RubyTL and Gra2MoL are reviewed below in the light of these aspects and with the perspective of time.

Concrete syntax

The concrete syntax required for the DSL is probably one of the main aspects that should be born in mind, because it is the front-end to the end-user. In this respect, if it must take on a certain shape (e.g., a well-known syntax for a certain target community) then the definition of an external DSL is generally recommended, since making the language internal will only be possible if the selected host language permits a suitable syntax. Languages with a non-intrusive syntax, such as Ruby, Smalltalk, Lisp or Haskell are therefore more likely to be used as host languages. However, the definition of an external DSL signifies that a grammar must be defined from scratch, which in most cases involves some extra work to define common language constructs such as expressions.

RubyTL did not require a very specialized syntax (beyond object-oriented manipulation in order to navigate models and write imperative code), or in other words, the host language syntax was suitable since the concrete syntax that we attained was sufficiently close to ATL and OCL, in the sense that only a few lexical variations were introduced (e.g., braces rather than parenthesis to denote the body of an iterator). However, in the case of Gra2MoL a concrete syntax close to XPath for the query language was required, as it was clear to us that an XPath-like syntax was suitable for the task that Gra2MoL was intended for. This kind of concrete syntax is in general difficult to achieve in an object-oriented language, and this was one of the reasons why we decided to implement Gra2MoL as an external DSL.

Abstract syntax

Using an abstract syntax model as the internal representation is recommended when the compilation or evaluation of the DSL is not straightforward, and it is not possible to use syntax-directed evaluation. This was the case of both RubyTL and Gra2MoL in which the rule evaluation was sufficiently complex to require an abstract syntax to guide the interpreter. As explained previously, in RubyTL the abstract syntax model only covered the transformation-specific parts (e.g., rules), which refer at runtime to Ruby code blocks. This can be seen as interleaving the Ruby abstract syntax with a domain-specific abstract syntax. The Gra2MoL abstract syntax, however, was bigger and more complex as it had to cover all language features (e.g., rules and query expressions) in order for it to be later fully evaluated by the language interpreter.

Executability

The executability aspect is closely related to the decision to implement a compiler or an interpreter. In both cases it is possible to make the DSL internal, but in our experience the maximum gain is obtained by

creating an interpreter since the runtime infrastructure of the host language can be reused. This is the case of RubyTL, in which the interpreter is very simple because it only deals with rule scheduling, while the rest of the execution is supported by Ruby itself. In this respect, there is a range of options when designing the language, from a so-called fluent API (Fowler, 2010) to more complex approaches like RubyTL. As regards Gra2MoL, there is no general-purpose language with built-in configurable support for XPath-like queries, and we could not therefore seek support from an existing execution infrastructure for Gra2MoL.

Optimizations

A typical limitation of internal DSLs is that it is difficult to implement domain-specific optimizations, and what is more, tweaking the host language to obtain certain syntax may involve performance penalties as is the case of RubyTL (e.g., meta-programming tricks which facilitate implementation at the cost of slowing down execution time). When fine-tuning is required, then an external DSL is the best option because developers can control the execution process. This is the case of Gra2MoL, in which we were able to boost the query execution performance. Related to this issue, a typical limitation of RubyTL transformations is that the abstract syntax model cannot be manipulated by another transformation (i.e., a higher-order transformation, also known as HOT) because it is a "mixed" abstract syntax model, as we have already explained. This has forced RubyTL users to move to ATL when they wish to use HOT techniques (including the developer of RubyTL himself).

When dealing with external DSLs, building an interpreter or a compiler requires a great implementation effort because developers must provide the execution semantics for each language construct. However, it allows developers to tune the execution process and to improve some features such as error control or performance. In our experience, the development of an interpreter is usually simpler than that of a compiler since developers do not have to define the translation to a low-level language, and the debugging and testing of the language is facilitated, although at the cost of a loss of performance.

Language extensions

Incorporating language extensions as they are demanded by language users could be thought of as being easier in internal DSLs (particularly when the host language is a dynamic language) than in external DSLs, where an extension could imply making in-depth changes to the language. However, in our experience this largely depends on the kind of extension and what parts of the language must be changed (i.e., syntax or semantics). From a general point of view, external DSLs are usually easier to adapt to new concrete syntax requirements whereas internal DSLs greatly depend on how well the extension fits into the host language. As an example, Gra2MoL was extended to support a kind of rule called "skip" which has its correspondence in RubyTL in the form of *one-to-many* mappings. From an implementation point of view, the extension was easier to implement in RubyTL because it was easy to integrate it into the core of the language (i.e., the semantics), leaving most of the evaluation to the Ruby interpreter itself. Instead, in Gra2MoL parts of the interpreter had to be rewritten. However, we were forced to make the concrete syntax fit into Ruby, while the Gra2MoL designer had the opportunity to choose the most appropriate one.

Integration

Integration with other tools can be a decisive factor, depending on the purpose of the DSL. In our experience this issue is not, in general, influenced by the internal/external dichotomy, but by the availability of the libraries required in a given programming language. This issue is clearly illustrated in the case of RubyTL. In the first versions we used an early version of RMOF, a Ruby meta-modeling framework, in order to be able to read/write models in XMI format. However, as the language became more stable, some users required a better integration with Ecore/EMF (the widest used modeling

framework). This forced us to practically re-implement RMOF to achieve a proper compatibility. Moreover, and despite our efforts, we never obtained a performance that was comparable to that of EMF in terms of execution time and memory consumption.

In some respects, the decision to implement Gra2MoL in Java was influenced by this experience, since it was clear that Gra2MoL should take advantage of the existing Eclipse modeling tools. In particular, Gra2MoL uses the EMF modeling framework to manage models and the CDO framework (CDO, 2012) to be able to store large models, since extracted models are normally large. Moreover, since it is written in Java (and probably as an external DSL) Gra2MoL has been proposed to become part of the MoDisco project (http://eclipse.org/MoDisco/). On the other hand, an alternative would have been to use JRuby (Ruby for the JVM), but at that time it was not as stable as it is now.

Another side of this aspect is integration with GPLs. In the case of an internal DSL, this is given by the very nature of the approach. In an external DSL, this functionality usually has to be created in an ad-hoc manner. As explained previously, Gra2MoL features a mechanism with which to add language extensions written in Java, which can be seen as a form of integration with a GPL. Some language workbenches, such as MPS, currently provide automatic support for this (Jetbrains, 2011).

Usability

Regarding usability, one important aspect to consider is IDE support. IDEs providing features such as syntax highlighting, code folding, auto-completion or cheat sheets are currently common for GPLs. An internal DSL not only inherits the host language's features, but it is also possible to take advantage of some features that are available in existing IDEs for the host language. For instance, features such as syntax highlighting and code folding are straightforward to reuse, while providing auto-completion based on the domain constructs is complicated to implement because it implies dealing with the whole grammar of the host language. On the other hand, external languages usually require the development of an IDE from scratch. However, it is currently possible to take advantage of tools such as such as xText (xText, 2011), TCS (Jouault et al., 2006) or Spoofax (Kats &Visser, 2010) to create IDEs for textual external languages including some advanced features (e.g., syntax highlighting, auto-completion and code folding).

RubyTL features an Eclipse-based IDE built on top of RDT (an extension of Eclipse for Ruby). We extended RDT with a functionality that was specific to RubyTL. Thus, with a limited effort we attained an editor with syntax highlighting, error and warning markers, program launchers, and a console with hyperlinks to navigate to source files when errors appeared. Figure 3 shows a screenshot of the RubyTL IDE. Building a similar environment for Gra2MoL proved too costly with similar resources since there were no mature tools to create IDEs at the moment of developing the IDE and everything had to be built from scratch. However, the Gra2MoL IDE does incorporate some language-specific features such as code completion, which would be too expensive to add to the RubyTL environment because it would require computing type information that is not statically available in a dynamic language like Ruby. Figure 4 shows a screenshot of the Gra2MoL IDE. The alternative would have been to alter the Ruby grammar to consider RubyTL specific constructs, but then it would no longer have been an internal DSL.

Figure 3. RubyTL IDE.

Figure 4. Gra2MoL IDE.

Target audience

Another aspect of usability is the target audience. As we have already discussed, if the syntax expected by the users cannot be emulated by the host language, then an external DSL is the only choice. Another

important aspect that must be considered before choosing the internal option is whether the target audience is used to the host language, and if not, whether they will reject the language because it implies learning a new general purpose language. In our experience, DSL users tend to perceive that an internal DSL is more complicated to learn because it implies learning a new general purpose language, even when it is sufficient to learn only a part of the host language. For instance, in RubyTL it is not necessary to learn about Ruby classes, instance variables, etc., and it is sufficient to learn the Ruby collection library to navigate models. However, we have witnessed that users tend to be reluctant to use RubyTL because of this. Thus, if it is possible the target audience will feel intimidated by the internal approach, an external DSL is recommended. On the other hand, the freedom offered by an external DSL to shape its syntax may also cause language users to request certain constructs that are common in other languages. For instance, Gra2MoL syntax was criticized by users who were used to defining model transformations with ATL and RubyTL because the `from` variable was not in the same place in the rule declaration.

Tooling

Finally, the availability and appropriateness of tools with which to create the DSL must be considered. Regular editors or environments are sufficient for the development of an internal DSL. In the case of external DSLs, it is possible to seek support from tools like xText or Spoofax, which in turn incorporate tools with which to define the corresponding IDE, as commented on previously. In the case of Gra2MoL, we could not use these tools since their maturity level was low. We therefore decided to implement the language from scratch and to later apply the bootstrapping process commented on before, which proved that the language could also be used to define external DSLs. However, if we decided to develop a new external DSL, we would use this kind of tools to ease the development process and the creation of the corresponding IDE

Evaluation and lessons learned

Our experience with RubyTL signified that when we decided to build Gra2MoL, we already had some insight into those situations in which making a DSL internal was not a good idea. We had learned that in an internal DSL the concrete syntax was somewhat limited, but over all it was a must for Gra2MoL to use EMF to deal with large models, and this framework is only available in Java. The fact is that the libraries required for a given DSL are more likely to be available for mainstream programming languages (e.g., Java, C++) than for languages that are suitable for internal DSLs (e.g., Ruby, Smalltalk, LISP). This situation is currently changing with languages built on top of the JVM and CLR such as JRuby (JRuby, 2012), Clojure (Clojure 2012) or Scala (Scala, 2012).

The comparison above provides further insight into which is the best choice in each case. It would also be interesting to compare the implementation effort. We have made this comparison using two kinds of metrics. We have first applied a set of metrics to the grammars of RubyTL and Gra2MoL with the aim of understanding their characteristics. We have then measured the number of lines of code (LOCs) involved in the implementation of each language.

We shall measure the languages by using classical metrics (Power &Malloy, 2004), namely: `TERM`, `VAR`, `MCC` and `HAL`, which will allow us to obtain a brief description of the grammar complexity. We shall additionally use the `LRS`, `LTPS`, `LAT/LRS` and `SS` metrics proposed in (Črepinšek, Kosar, Mernik, Cervelle, Forax & Roussel, 2010), which will allow us to provide a better characterization of each language. For the sake of concreteness, we shall not detail each metric, but simply discuss their meaning with regard to RubyTL and Gra2MoL. Interestingly, there is no actual RubyTL grammar but as an internal DSL it programmatically inherits and “extends” the one from Ruby (i.e., grammar productions are not added by using “keyword-methods” as has been explained). Therefore, we have applied the metrics to Ruby, but for analysis purposes we have also manually enriched the Ruby grammar to consider RubyTL constructs.

Table 1 shows the results obtained, along with a brief explanation of each metric. The RubyTL column contains the values for the enriched Ruby grammar, and also indicates the variation regarding the Ruby values in a percentage. The results for Gra2MoL clearly denote its condition as a DSL: low values in HAL, LRS and LTPS metrics. On the other hand, the interpretation of the values for RubyTL is actually difficult because, as a Ruby internal DSL, the language inherits the characteristics from this host language. However, we can analyze how RubyTL alters the metric results. For instance, although the complexity of the Ruby grammar is slightly increased (see HAL and LRS values), the resulting internal language is actually easier to learn according to the LAT/LRS value. Upon considering the different nature of both DSLs, if both languages are compared, Gra2MoL is clearly simpler than RubyTL (see HAL and LRS values). However, the low value of LAT/LRS, along with a high verbosity (see SS value), denotes that RubyTL is easier to learn than Gra2MoL, whose query language format may influence this metric.

Metric	Explanation	Gra2MoL	Ruby	RubyTL
TERM	Number of grammar terminals	71	88	108 (+23%)
VAR	Number of grammar non-terminals	32	83	99 (+19 %)
MCC	Mcabe Cyclomatic Complexity: Effort for grammar testing and more potential parsing conflicts	3.4	2.61	2.37 (-9%)
HAL	Designer effort to understand the grammar	36.86	54.44	62.69 (+15%)
LRS	Grammar complexity independent of its size	5	13474	14.21 (+5%)
LTPS	Indicate language type. GPL > 1000	334	3200	3521 (+22%)
LAT/LRS	Facility to learn the language. Lower value is easier to learn.	0.28	0.26	0.21 (-19%)
SS	Verbosity of the language	1	1.47	1.8 (+22%)

Table 1.Metric results for Gra2MoL, Ruby and RubyTL

Table 2 summarizes the LOCs written to implement the concrete syntax, the core of the interpreter, and support libraries (e.g., integration with the modeling framework), and Figure 4 shows the percentage of implementation devoted to each of the components. Please bear in mind that it is expected that the same functionality in Ruby takes a few less LOCs.

	RubyTL	Gra2MoL
Concrete syntax	331	781 (194 + 587)
Interpreter	1489	5133
Rule engine	1127	741
Expressions	362	4392
Model manager	737	933
Modeling framework	2187	–
Total	4744	6847

Table 2. Lines of code (without comments and blank lines) involved in RubyTL and Gra2MoL

Figure 5. LOC distribution for each DSL component between RubyTL and Gra2MoL.

As expected, the effort involved in defining the concrete syntax is less for RubyTL, and more so if we consider that the LOCs for RubyTL are just plain Ruby code, while for Gra2MoL it involves the grammar (194 LOCs) and the Gra2MoL bootstrapping transformation (587 LOCs). The interpreter is split into two parts: the rule engine and the expression evaluator. The rule engine has a similar complexity in both RubyTL and Gra2MoL (there are more LOCs for RubyTL because it includes a modularity mechanism that is not present in Gra2MoL (Sánchez Cuadrado & García Molina, 2010). The expression evaluator requires almost no code in RubyTL since it is an internal DSL, only some tweaks to overload the assignment operator for bindings. However, it involves much more effort in Gra2MoL as it was

developed from scratch. The model managers perform similar tasks in both cases, so they have similar LOCs.

As explained previously, RubyTL necessitated the creation of a modeling framework that was compatible with Ecore/EMF in Ruby, called RMOF. The effort of creating this framework is comparable to that of creating RubyTL itself. What is more, it could not be reused for Gra2MoL owing to the lack of efficiency with regard to EMF.

A first conclusion that can be drawn from these measures, is that the cost of building RubyTL is approximately half that of Gra2MoL, with the additional benefit that RubyTL has more features than Gra2MoL, in the sense that it provides the possibility of using Ruby features seamlessly (e.g., imperative constructs, regular expressions). However, if RMOF is taken into account then the total effort is similar. If we had realized sooner that libraries and integration were such important issues, then we would have probably chosen to implement RubyTL as an external DSL. On the other hand, it is true that we were able to modify and experiment with RubyTL really quickly, which provided benefits since it allowed us to devise new transformation mechanisms (Sánchez Cuadrado & García Molina, 2010).

Another conclusion is that an internal approach makes more sense if the runtime infrastructure of the host language can be reused, as was the case of RubyTL. The cost of building an internal DSL that does not delegate on the host language for execution (i.e., building a complete abstract syntax model) would be equivalent to the external approach. As can be observed, the LOCs of both rule engines and model managers are alike, but the difference is that most of the implementation effort of Gra2MoL was in the query engine. The RubyTL interpreter was, in contrast, much simpler because it only dealt with the rule engine.

With regard to IDE support, if having domain-specific assistance in the DSL editor is important then the external DSL approach is the best choice, and more so when taking into account the existence of tools like xText, TCS or Spoofax.

Finally, regarding the target audience and the adoption of the language, it is our opinion that it is more difficult to engage users in an internal DSL than an external DSL when they do not already know the host language. This can be an insurmountable obstacle to the adoption of the language. Thus, if your target audience does not have some expertise in the host language, then it is better to choose the external approach.

RELATED WORK

The comparison presented in this chapter can be related to other works that report on DSL implementation and design techniques, in addition to works that describe DSLs for writing model transformations with similarities to RubyTL and Gra2MoL.

With regard to DSL techniques, Spinellis (Spinellis, 2001) identifies several design patterns that can be applied in the design and implementation of textual DSLs. These patterns tackle some recurrent problems that occur when dealing with DSLs, such as composition and specialization, and several DSL examples are given for each pattern. This work was extended in (Mernik, Heering & Sloane, 2005), in which the patterns are organized according to the phases in the DSL development process: decision, analysis, design and implementation. In (Czarnecki, Donnell & Taha, 2004) static meta-programming techniques applied to building internal DSLs in C++, OCaml, and Haskell are compared. In (Fowler, 2010) many topics related to the design and implementation of both internal and external DSLs are discussed. Finally, it is worth noting that in some works such as (Hudak, 1996) internal DSLs are referred to as embedded DSLs. The term embedded DSL is currently also used to refer to those DSLs created with language workbenches

that provide seamless composition of heterogeneous DSLs (Jetbrains, 2011; Kats & Visser, 2010), that is, heterogeneous embedding. In contrast an internal DSL is a homogenous embedding, as the host language itself is used for implementing DSL.

Some works have carried out empirical studies for different implementation approaches. In (Kosar, Martinez-Lopez, Barrientos & Mernik, 2008), the same textual DSL is implemented with several approaches. Some conclusions are of interest to this chapter: i) When using effective lines of code (eLOC) as a metric, the internal approach was the most efficient way in which to implement DSLs; ii) the original notation was hard to achieve in the case of the internal approach and iii) error reporting and debugging was unsatisfactory for the internal approach. Another kind of experiment has been carried out in (Hermann, Pinzger & Deursen, 2009), in which the aim was to find the factors contributing to the success of a DSL, and the authors elaborated a survey to measure the success of the ACA-NET DSL amongst users of 30 projects all around the world. Factors such as usability, learnability, expressiveness, and reduction of the development costs are presented as factors for DSL success, and some lessons learned are discussed.

On the other hand, in the last few years several DSLs have been created for purposes similar to that of RubyTL and Gra2MoL. Some of these are reviewed as follows in order to highlight the variety of decisions that can be made. As acknowledged by (Czarnecki & Helsen, 2006) a diversity of model-transformation languages currently exists. In particular, ATL (Jouault, Allilaire, Bézivin & Kurtev, 2008) is the most similar to RubyTL with regard to its semantics. It is an external DSL, whose concrete syntax has been built with TCS (Jouault et al., 2006). The executability aspect is covered with a compiler from ATL to a dedicated virtual machine. It also features an IDE that is similar to RubyTL, but auto-completion has recently been added to it. A comparison between the implementations of ATL and RubyTL can be found in (Sanchez Cuadrado & García Molina, 2007). SiTra (Akehurst, Bordbar, Evans, Howells & McDonald-Maier, 2007) is a model transformation language built as an internal DSL in Java. It has mechanisms for defining rules that are implemented as Java classes. However, as Java does not have a non-intrusive syntax, it could be considered to be simply an API. A similar approach is taken for program transformation in Kiama (Sloane, Kats &Visser, 2011). However, in this case the host language is Scala, which permits a more flexible syntax. It is worth noting that neither SiTra nor Kiama provides an adapted IDE or specialized error control support.

Gra2MoL is related to languages that deal with grammar-based artifacts. Examples of those languages are the Silver attribute grammar system (Wyk, Bodin & Gao, 2010) and TXL (Cordy, 2006). In both cases, a command-line compiler is the tool front-end, and no IDE support is available. The LISA system is an example of grammar system that assists in generating DSLs as well as their associated IDEs (Henriques, et al., 2005). The Spoofax (Kats &Visser, 2010) environment is a language workbench for Eclipse that uses the Stratego program transformation system as its core. In addition to basic features such as syntax highlighting, it enables more complex features such as auto-completion.

CONCLUSION

In this chapter we have discussed the advantages and disadvantages of making a DSL internal or external, with the aim of providing developers who have to confront this question with guidance when beginning the development of a DSL. To this end, we have compared RubyTL and Gra2MoL, two transformation languages built using the internal and external approach respectively.

Both RubyTL and Gra2MoL have now been in use for several years, and some of the expectations of them have been satisfied, while others have not. We believe that choosing the internal or the external implementation technique has had a direct influence. To summarize the lessons learned during this time, an internal DSL is a good choice when the host language can support the DSL syntax seamlessly, there

are no st.rong performance constraints, the host language runtime infrastructure can be heavily reused, and the target audience knows the host language or, at least, is not reluctant to learn it. In the other cases, we believe that an external DSL will be a better option.

Acknowledgements
This work has been supported by the Spanish government through the TIN2009-11555 project. We also thank the reviewers for their insightful comments.

REFERENCES

Aho, A.V., & Ullman, J.D. (1977). *Principles of Compiler Design.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Akehurst, D., Bordbar, B., Evans, M., Howells, W., & McDonald-Maier, K. (2007). SiTra: Simple Transformations in Java. In Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007 (pp. 351-364). Lecture Notes in Computer Science, vol. 4735, Springer.

Bentley, J. (1986). Programming pearls: little languages. *Communications of the ACM, 29*(8), 711-721.

Cánovas, J.L., & García-Molina, J. (2009). A Domain Specific Language for Extracting Models in Software Modernization. In R. F. Paige, A. Hartman, A. Rensink (Eds.), Model Driven Architecture - Foundations and Applications, 5th European Conference, ECMDA-FA 2009 (pp. 82-97). Lecture Notes in Computer Science, vol. 5562, Springer.

CDO (2011). CDO Framework. Retrieved March 23, 2012, from http://www.eclipse.org/cdo/

Clojure (2012). Clojure language. Retrieved March 23, 2012, from http://www.clojure.org

Cook, S., Jones, G., Kent, S., & Wills, A.C. (2007). *Domain-Specific Development with Visual Studio DSL Tools*, Boston, MA, USA. Pearson Education, Inc.

Cordy, J.R. (2006). The TXL Source Transformation Language, *Science of Computer Programming 61*(3), 190-210.

Črepinšek, M., Kosar, T., Mernik, M., Cervelle, J., Forax, R., & Roussel G. (2010). On Automata and Language Based Grammar Metrics. *Journal on Computer Science and Information Systems, 7*(2), 310-329.

Czarnecki, K., Donnell, J. O., & Taha, W. (2003). DSL Implementation in MetaOCaml, Template Haskell, and C++. In C. Lengauer, D. S. Batory, C. Consel, M. Odersky (Eds.), Domain-Specific Program Generation, International Seminar (pp. 1-22), Lecture Notes in Computer Science, vol. 3016, Springer.

Czarnecki, K., & Helsen, S. (2006). Feature-based survey of model transformation approaches. IBM System Journal, *45*(3), 621-645.

Diaz, O., Puente, G., Cánovas, J.L., & García Molina, J. (2012). Harvesting models from web 2.0 databases. *Software and Systems Modeling*, DOI: 10.1007/s10270-011-0194-z, available online from http://www.springerlink.com/content/02703489388027g2/.

Dmitriev, S. (2004). Language Oriented Programming: The Next Programming Paradigm. Retrieved October 5, 2011, from http://www.omg.org/cgi-bin/doc?omg/03-06-01

Fowler, M. (2010). *Domain-Specific Languages.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Greenfield, J., Short, K., Cook, S., & Kent, S. (2004). *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Indianapolis, Indiana, USA: Wiley Publishing.

Henriques, P. R., Varando Pereira, M. J., Mernik, M., Lenic, M, Gray, J., & Wu, H. (2005). Automatic generation of language-based tools using the LISA system. *Software IEE Proceedings. 152* (2), 54-69.

Hermans, F., Pinzger, M., & van Deursen, A. (2009), Domain-Specific Languages in Practice: A User Study on the Success Factors. Report TUD-SERG-2009-013. Delft University of Technology. Retrieved March, 28, 2012, from http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2009-013.pdf

Hudak, P. (1996). Building domain-specific embedded languages. *ACM Computing Surveys 28*(4es), 196.

JetBrains (2011). MPS. Retrieved March, 28, 2012, from http://www.jetbrains.com/mps

Jouault, F., Allilaire, F., Bézivin, J., & Kurtev, I. (2008) ATL: A model transformation tool. *Science of Computer Programming*, *72*(1), 31-39.

Jouault, F. and Bézivin, J., & Kurtev, I. (2006). TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In S. Jarzabek, D. C. Schmidt, T.L. Veldhuizen (Eds.), Generative Programming and Component Engineering, 5th International Conference, GPCE, 2006 (pp. 249-254). ACM

JRuby (2012). JRuby. Retrieved March, 28, 2012, from http://www.jruby.com

Kats, C.L., & Visser, E. (2010). The spoofax language workbench: rules for declarative specification of languages and IDEs. In W. R. Cook, S. Clarke, M. C. Rinard (Eds.), Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, 2010 (pp. 444-463). ACM.

Kelly, S., & Tolvanen, J. P. (2008). *Domain-Specific Modeling: Enabling full code generation.* Hoboken, New Jersey, USA: John Wiley & Sons, Inc.

Kleppe, A. (2008). *Software Language Engineering*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Kosar, T., Martinez López, P. E., Barrientos, P.A., & Mernik, M. (2008). A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology, 50*(5), 390-405.

Kosar, T., Mernik, M., & Carver, J. (2012). Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical Software Engineering*, *17*(3), 276-304.

Ledeczi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., & Karsai, G. (2001). Composing Domain-Specific Design Environments. *Computer*, 34(11), 44-55.

Mernik, M., Heering, J., & Sloane, A. M. (2005).When and how to develop domain-specific languages. *ACM Computing Surveys, 37*(4), 316-344.

Object Management Group. (2001). MDA Guide. Retrieved October 5, 2011, from http://www.omg.org/cgi-bin/doc?omg/03-06-01

Power, J.F., & Malloy, J.F. (2004). A metrics suite for grammar-based software. *Journal of Software Maintenance and Evolution: Research and Practice, 16*(6), 405–426.

Sánchez Cuadrado, J., & García Molina, J. (2007). Building Domain-Specific Languages for Model-Driven Development. *IEEE Software, 24*(5), 48-56.

Sánchez Cuadrado, J., & García Molina, J. (2009). A Model-Based Approach to Families of Embedded Domain Specific Languages. *IEEE Transactions on Software Engineering, 25*(6), 825-840.

Sánchez Cuadrado, J., & García Molina, J. (2010). Modularization of model transformations through a phasing mechanism. *Software and Systems Modeling, 8*(3), 325-345.

Sánchez Cuadrado, J.,García Molina, J., & Menárguez, M. (2006). RubyTL: A Practical, Extensible Transformation Language. In A. Rensink, J. Warmer (Eds.), Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA, 2006 (pp. 158-172). Lecture Notes in Computer Science, vol. 4066, Springer.

Scala (2012). Scala language. Retrieved March, 28, 2012, from http://www.scala-lang.com

Sloane, A.M., Kats L.C.L., & Visser, E. (2011). A pure embedding of attribute grammars. *Science of Computer Programming,* Available online as http://dx.doi.org/10.1016/j.scico.2011.11.005.

Voelter, M. (2008). MD* Best Practices. *Journal of Object Technology, 8*(6), 79-102.

Weiss, D.M., & Lai, C.T.R. (1999). *Software product-line engineering: a family-based software development process.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Wyk, V. E., Bodin, D., & Gao, J. (2010). Silver: an Extensible Attribute Grammar System. *Science of Computer Programming, 75*(2), 39-54.

XPath (2012). XML Path Language (XPath) 2.0. Retrieved March, 28, 2012, from http://www.w3.org/TR/xpath20/

xText (2011). xText 2.0. Retrieved March, 28, 2012, from http://www.xtext.org

KEY TERMS AND DEFINITIONS

Domain-Specific Language (DSL): A language specifically tailored to address a problem or a task in a particular application domain.

Concrete syntax: The notation that the users of a language are provided with in order to develop programs or specifications.

Abstract syntax: The construction rules of a language without taking notational details into account, that is, the valid constructs and their composition rules.

External DSL: An approach with which to implement a DSL, in which the language is built from scratch so that it has a custom made concrete syntax and a specific infrastructure.

Internal DSL: An approach with which to implement a DSL, in which the language is built on top of another language, the host language, reusing its infrastructure.

Model-Driven Engineering: A family of software developing paradigms which promote the pervasive use of models in the software development cycle. Meta-modeling and model transformation are key elements of this approach.

Meta-modeling: Construction and support to the elements that allow models that represent a certain system or domain of interest to be described.

Model transformation: The manipulation of model(s) that conform(s) to a particular meta-model. It includes, among others, in-place transformations, model-to-model transformations or text-to-model transformation.

FURTHER READING

Aβmann, U., & Sloane, A. (Eds.) (2012). *Software Language Engineering, Proceedings of the Fourth International Conference, SLE 2011*. Lecture Notes in Computer Science, vol. 6940, Springer.

Batory, D. S., Johnson, C., MacDonald, B., & von Heeder, D. (2002). Achieving extensibility through product-lines and domain-specific languages: a case study. *ACM Transactions on Software Engineering Methodology*, 11(2), 191-214.

van den Brand, M., Gasevic, D., & Gray, J. (Eds.) (2010). *Software Language Engineering, Proceedings of the Second International Conference, SLE 2009*. Lecture Notes in Computer Science, vol. 5969, Springer.

Bravenboer, M., de Groot, R., & Visser, E. (2006). MetaBorg in Action: Examples of Domain-Specific Language Embedding and Assimilation Using Stratego/XT. In R. Lammel, J. Saraiva, and J. Visser (Eds.), Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005 (pp. 297–311). Lecture Notes in Computer Science, vol. 4143, Springer.

Clark, T., Sammut P., & Willans, J. (2004). *Applied Metamodelling: A Foundation for Language Driven Development*. Retrieved October 5, 2011 from http://eprints.mdx.ac.uk/6060/1/Clark-Applied_Metamodelling_(Second_Edition)%5B1%5D.pdf

Cleaveland, J. C. (1988). Building application generators. *IEEE Software*, *5*(4) 25-33.

Czarnecki, K., & Eisenecker, U. (2000). *Generative Programming: Methods, Techniques and Applications*. Boston, MA, USA: Addison-Wesley Co.

van Deursen, A., & Klint, P. (1998). Little languages: little maintenance? *Journal of Software Maintenance: Research and Practice, 10*(2), 75-92.

van Deursen, A. Klint, P., & Visser, J. (2000). Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices, 35*(6), 26-36.

Favre, J.M, Gaševic, D., Lammel, R., & Winter, A. (eds.) (2009). *IEEE Transactions on Software Engineering*. Special issue on software language engineering. *35*(6).

Gaševic, D., Lämmel, R., & Van Wyk, E. (2009). *Software Language Engineering, Proceedings of the First International Conference, SLE 2008*. Lecture Notes in Computer Science, *vol.* 5452, Springer.

Ghosh, D. (2010). *DSLs in Action*. Stamford, Connecticut, USA. Manning Publications Co.

Gronback, R. C. (2009). *Eclipse Modeling Project. A Domain-Specific Language (DSL) Toolkit*. Boston, MA, USA: Pearson Education Inc.

de Groot, R. (2005). Implementation of the Java-Swul language a domain-specific language for the SWING API embedded in Java. Retrieved March, 28, 2012, from http://www.program-transformation.org/pub/Stratego/Java-Swul/swul-article.pdf

Kelly, S., & Pohjonen, R. (2009). Worst Practices for Domain-Specific Modeling. *IEEE Software*, *26*(4), 22-29.

Krueger, C.W, (1992). Software reuse. *ACM Computing Surveys, 24*(2), 131-183.

Kurtev, I., Bezivin, J., Jouault, F., & Valduriez, P. (2006). Model-based DSL frameworks. In P.L. Tarr, W.R. Cook (Eds.), Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, 2006 (pp. 602-616). ACM.

Malloy, B.A., Staab, S., & van den Brand, M. (Eds.) (2011). *Software Language Engineering, Proceedings of the Third International Conference, SLE 2010*. Lecture Notes in Computer Science, vol. 6563, Springer.

Parr, T. (2010). *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Lewisville/TX, USA: The Pragmatic Bookshelf.

Selic, B. (2008). Personal reflections on automation, programming culture, and model-based software engineering. *Automated Software Engineering*. *15*(3), 379-391.

Spinellis, D. (2001). Notable design patterns for domain specific languages, *Journal of Systems and Software*, *56*(1), 91–99.

Steinberg, D., Budinsky, F., Paternostro, M., & Merks, E., (2008). *Eclipse Modeling Framework*. Boston, MA, USA: Pearson Education Inc.

Thomas D., & Hunt, A. (2000). *Programming Ruby: The pragmatic programmer's guide*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Voelter, M. (2010). Embedded software development with projectional language workbenches. In D.C. Petriu, N. Rouquette, Ø. Haugen (Eds.), Model Driven Engineering Languages and Systems, 13th International Conference, MODELS 2010 (pp. 32-46). Lecture Notes in Computer Science, vol. 6394, Springer.

Wile, D.S., & Ramming, J.C. (eds) (1999). *IEEE Transactions on Software Engineering*. Special issue on Domain-Specific Languages. *25*(3).